Problem 1: Two Sum

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

int main() {
  int n, target;
  cin >> n >> target;
  unordered_map<int, int> hashMap;
  int i = 0;
  while (i < n) {
    int curNum = 0;
    cin >> curNum;
    int toFind = target - curNum;
    if (hashMap.find(toFind) == hashMap.end()) {
      hashMap.insert({curNum, i++});
    } else {
      cout << hashMap.at(toFind) + 1 << ' ' << i + 1 << endl;
      return 0;
    }
  }
  return 1;
}
```

We need to traverse the array to that pair. Instead of scanning the whole array when we reach one number, we can just limit our scope in its previous elements, which can avoid the duplicated pair formation, that is (x, y) and (y, x). Though this approach save some time, it is still O(n^2). As we need to traverse the array at least once to cover all possible circumstances, the key is to shorten the time of finding counter part of a number in its previous. It is a good way to make use of `unordered_map`, whose underlying structure is hash map. According to cppreference.com, the average time complexity is O(1) to make access to an element. We can apply this trait to boost our find counterpart procedure. We may store the previous number into the map as `<number, index>` pair, and directly try to access it by calculate `target-currentNum`, which is the key of out wanted pair.

Problem 2: Inversion Number

```cpp
#include <iostream>
#include <vector>
using namespace std;

unsigned long long countInversions(vector<int>& nums) {
  auto size = nums.size();
  if (size == 1) {
    return 0;
  } else {

    // divide and count
    vector<int> former(nums.begin(), nums.begin() + size / 2);
    vector<int> latter(nums.begin() + size / 2, nums.end());
    auto countF = countInversions(former);
    auto countL = countInversions(latter);

    // merge and count
    unsigned long long i = 0, j = 0, k = 0, res = 0;
    while (k < size) {
      if (i < former.size() && j < latter.size()) {
        if (former[i] <= latter[j]) {
          nums[k++] = former[i++];
        } else {
          res += (former.size() - i);
          nums[k++] = latter[j++];
        }
      } else if (i == former.size()) {
        nums[k++] = latter[j++];
      } else {
        nums[k++] = former[i++];
      }
    }
    return res + countF + countL;
  }
}

int main() {
  int size, k;
  cin >> size;
  vector<int> nums;
  for (int i = 0; i < size; i++) {
    cin >> k;
    nums.push_back(k);
  }
  cout << countInversions(nums) << endl;
  return 0;
}
```

We can modify the merge sort to find the inversion number, which is the same approach as in Assignment 1. The inversion count consists of three parts: the inversion pairs in the first-half array, the ones in the second-half array and the ones across there two part, that is, one element in the pair is in the first-half array while the other one is in the second-half array. The first two part can be counted in the divide and conquer approach like merge sort. But we also need to add some extra instruction in the merge procedure to count the third part. By observation, we can easily find that if a number in the second half less than the one in first half in merge process, we can form as many of inversion pairs as the remaining non-iterated number in first half. We also need to pay attention to that we need a bigger type store the result instead of `int`, which may lead to overflow.

Problem 3: Line Arrangement

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

// used to store adjacent numbers
struct lr {
  int left = 0;
  int right = 0;
  lr() {}
  lr(int left) { this->left = left; }
  lr(int left, int right) {
    this->left = left;
    this->right = right;
  }
};

int main() {
  int count;
  cin >> count;
  unordered_map<int, lr> map{};
  map.insert({1, lr()});

  // add part
  for (int i = 2; i <= count; i++) {
    int x, p;
    cin >> x >> p;
    if (p) {
      if (map[x].right) {
        map.insert({i, lr(x, map[x].right)});
        map[map[x].right].left = i;
      } else {
        map.insert({i, lr(x)});
      }
      map[x].right = i;
    } else {
      if (map[x].left) {
        map.insert({i, lr(map[x].left, x)});
        map[map[x].left].right = i;
      } else {
        map.insert({i, lr(0, x)});
      }
      map[x].left = i;
    }
  }

  // remove part
  cin >> count;
  while (count != 0) {
    int x;
    cin >> x;
```

```
      if (map[x].left != 0 && map[x].right != 0) {
        map[map[x].left].right = map[x].right;
        map[map[x].right].left = map[x].left;
      } else if (map[x].left != 0) {
        map[map[x].left].right = 0;
      } else {
        map[map[x].right].left = 0;
      }
      map.erase(x);
      count--;
    }

    // find the first one
    int num = map.begin()->first;
    while (map[num].left != 0) {
      num = map[num].left;
    }
    while (num != 0) {
      cout << num << ' ';
      num = map[num].right;
    }
    return 0;
  }
```

It is clear that we need at least O(n) + O(m) time for construct n person into the data structure and then remove m of them. We can boost our program by reducing the searching time, similar as the Problem 1. Therefore, we can use a hash map to achieve this goal. But the hash map does not contain any position information. We can customize a data structure to store the adjacent numbers and store it into the hash map as `<number, adjacent pair>`. So we can make access to any element in average time complexity O(1) but also keep their position information.